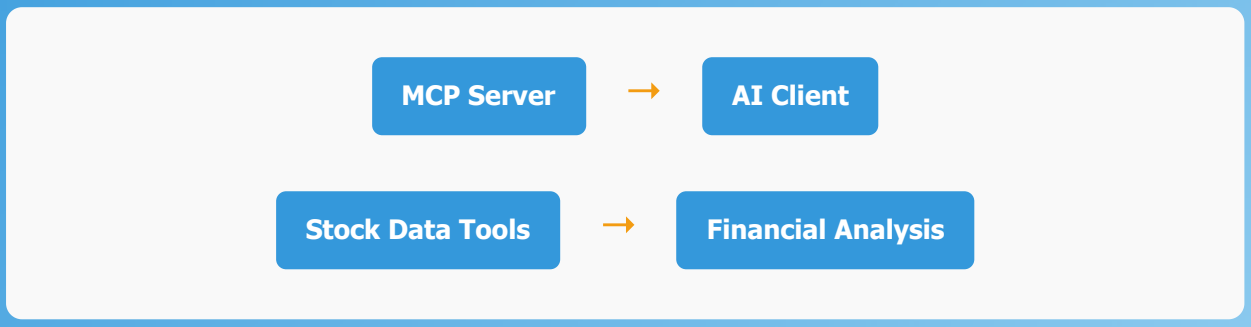
Understanding MCP Servers

A Comprehensive Guide to Model Context Protocol with Stock Server Example



Created by
Dr. Anish Roychowdhury
Plaksha University

Version 1 : September 14, 2025

What This Example Demonstrates

This guide uses a **Stock Price Server** as a practical example to teach MCP concepts. The stock server demonstrates how to:

- Fetch real-time stock data from Yahoo Finance API
- Create interactive financial charts and visualizations
- Compare stock performance across different time periods
- Expose stock information through MCP resources
- Handle date parsing, error management, and data validation

By the end of this guide, you'll have a fully functional MCP server that can analyze stock market data and understand the core principles that apply to any MCP implementation.

Table of Contents

1. Introduction to MCP Servers

What is MCP?

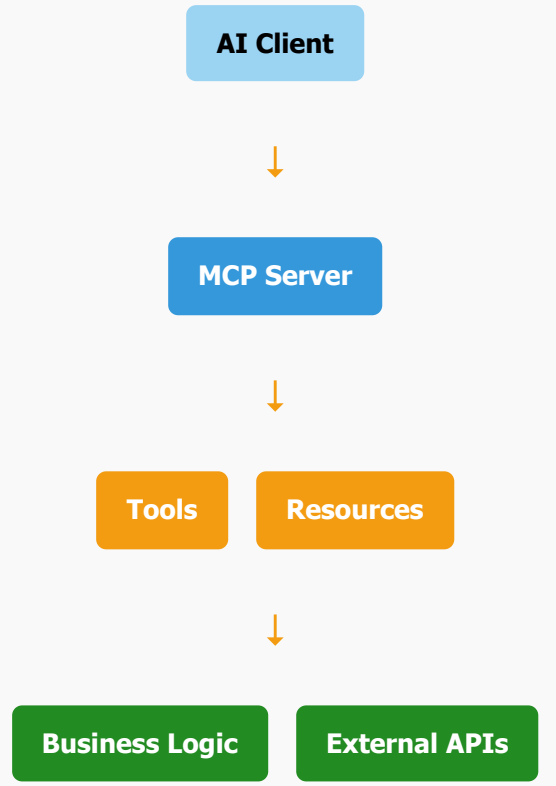
The Model Context Protocol (MCP) is a communication standard that allows AI models to securely connect to external tools and data sources. It provides a standardized way for AI assistants to access databases, APIs, file systems, and other resources.

MCP servers act as bridges between AI models and external systems. They expose tools and resources that AI assistants can use to perform tasks beyond their built-in capabilities.

Key Components of MCP

1. **Tools**: Functions that the AI can call to perform actions
2. **Resources**: Data sources that the AI can read from
3. **Prompts**: Reusable prompt templates
4. **Transport**: Communication layer (stdio, HTTP, WebSocket)

2. MCP Server Architecture



Communication: JSON-RPC over stdin/stdout

3. Creating Your First MCP Server: Stock Price Server

Now let's build a stock price MCP server step by step.

Project Structure

File Organization

```
stock_mcp_server/
├── stock_server.py       # Main MCP server
├── requirements.txt      # Dependencies
└── README.md            # Documentation
```

Installation and Setup

Step 1: Install Dependencies

```
# Install Required Packages
pip install fastmcp yfinance matplotlib pandas numpy
```

Tool Implementation Breakdown

Date Parsing Utility

```
def parse_date(date_str: str) -> str:
    """Convert mmddyyyy format to yyyy-mm-dd format for yfinance"""
    month = date_str[:2]
    day = date_str[2:4]
    year = date_str[4:]
    dt = datetime(int(year), int(month), int(day))
    return dt.strftime('%Y-%m-%d')
```

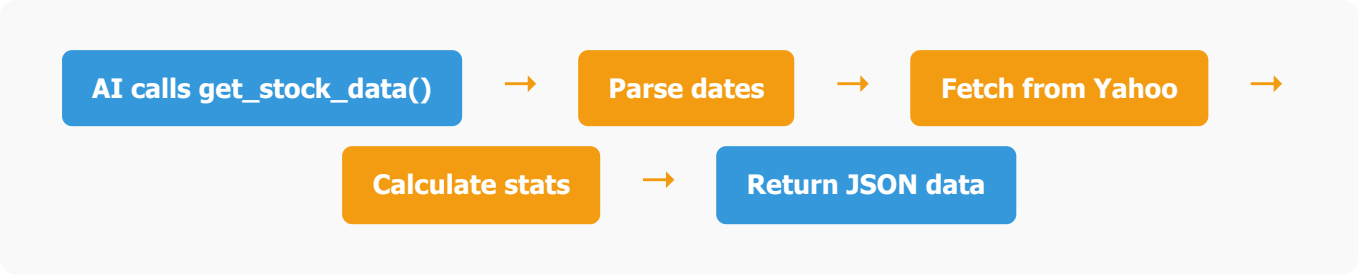
Stock Data Tool

```
@mcp.tool()
async def get_stock_data(ticker: str, name: str, start_date: str, end_date: str) -> dict:
    """Fetch stock price data from Yahoo Finance."""
    start_formatted = parse_date(start_date)
    end_formatted = parse_date(end_date)

    stock = yf.Ticker(ticker)
    data = stock.history(start=start_formatted, end=end_formatted)
```

```
    # Calculate statistics
    stats = {
        "ticker": ticker,
        "opening_price": round(data['Open'].iloc[0], 2),
        "closing_price": round(data['Close'].iloc[-1], 2),
        "percentage_change": round(((data['Close'].iloc[-1] / data['Open'].iloc[0]) - 1) * 100, 2)
    }
    return stats
```

4. Understanding the Tool Chain



5. Running Your MCP Server

Command Line Usage

Step 1: Start the Server

```
# Start MCP Server
python stock_server.py
```

Step 2: Test Tool Calls

The server accepts JSON-RPC messages via stdin/stdout. Example tool call:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/call",
  "params": {
    "name": "get_stock_data",
    "arguments": {
      "ticker": "AAPL",
      "name": "Apple Analysis",
      "start_date": "01012023",
      "end_date": "12312023"
    }
  }
}
```

6. Advanced Features

Resource Endpoints

Resources provide read-only access to data:

```
@mcp.resource("stock://data/{ticker}")
async def read_stock_resource(ticker: str) -> str:
    """Resource endpoint to get current stock information"""
    stock = yf.Ticker(ticker)
    info = stock.info
    return f"Company: {info.get('longName', 'N/A')}"
```

Error Handling Best Practices

Error Handling Guidelines

- Always validate input parameters
- Return descriptive error messages
- Handle network failures gracefully
- Log errors for debugging

7. Best Practices for MCP Server Development

1. **Clear Tool Documentation**
   - Write descriptive docstrings
   - Specify parameter types and formats
   - Include usage examples
2. **Input Validation**
   - Validate all input parameters
   - Handle edge cases gracefully
   - Provide meaningful error messages
3. **Error Handling**
   - Use try-catch blocks appropriately
   - Return structured error responses
   - Log errors for debugging
4. **Performance Optimization**
   - Cache frequently accessed data
   - Use async/await for I/O operations
   - Implement request rate limiting

8. Practical Implementation Guide

Setting Up Your Development Environment

Step 1: Project Structure

Create a proper project structure for maintainability:

```
mcp_stock_project/
├── stock_server.py           # Main MCP server
├── example_client.py         # Test client
├── setup_and_test.py        # Setup script
├── debug_and_troubleshoot.py # Debug utilities
├── requirements.txt          # Dependencies
├── test_config.json         # Test configuration
├── USAGE.md                 # Documentation
└── logs/                    # Log files
```

Step 2: Installation and Verification

Run the setup script to ensure everything works:

```
# Install and verify
python setup_and_test.py

# Run debugging tests
python debug_and_troubleshoot.py

# Test the server
python example_client.py

# Interactive testing
python example_client.py interactive
```

9. Common Issues and Solutions

Troubleshooting Guide

The debugging script helps identify and fix common issues automatically.

Import Errors

```
# Common fix for FastMCP import issues
pip install --upgrade fastmcp

# For matplotlib backend issues
import matplotlib
matplotlib.use('Agg')  # Non-interactive backend
import matplotlib.pyplot as plt
```

Yahoo Finance Rate Limiting

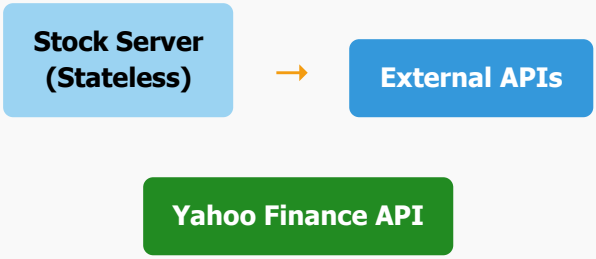
```
import time
from functools import wraps

def rate_limit(calls_per_second=1):
    def decorator(func):
        last_called = [0.0]
        @wraps(func)
        def wrapper(*args, **kwargs):
            elapsed = time.time() - last_called[0]
            left_to_wait = 1.0 / calls_per_second - elapsed
            if left_to_wait > 0:
                time.sleep(left_to_wait)
            ret = func(*args, **kwargs)
            last_called[0] = time.time()
            return ret
        return wrapper
    return decorator

@rate_limit(0.5)  # Max 1 call per 2 seconds
def get_share_price(symbol: str) -> float:
    # Your Yahoo Finance call here
    pass
```

10. Advanced MCP Concepts

State Management Patterns



Stateless Servers: Like our stock server, fetch data on-demand from external sources

Stateful Servers: Maintain persistent state in databases or memory

Error Handling Strategies

```
@mcp.tool()
async def robust_stock_data(ticker: str, start_date: str, end_date: str) -> dict:
    """Robust stock data fetching with comprehensive error handling"""
    try:
        # Input validation
        if not ticker or len(ticker) < 1:
            return {"error": "Invalid ticker symbol", "code": "INVALID_INPUT"}

        # Date validation
        try:
            start_formatted = parse_date(start_date)
            end_formatted = parse_date(end_date)
        except ValueError as e:
            return {"error": str(e), "code": "INVALID_DATE"}

        # API call with retry logic
        for attempt in range(3):
            try:
                stock = yf.Ticker(ticker)
                data = stock.history(start=start_formatted, end=end_formatted)

                if data.empty:
                    return {
                        "error": f"No data available for {ticker}",
                        "code": "NO_DATA",
                        "suggestions": ["Check ticker symbol", "Try different date range"]
                    }

                return {"data": process_data(data), "success": True}

            except ConnectionError:
                if attempt < 2:
                    await asyncio.sleep(2 ** attempt)  # Exponential backoff
                    continue
                return {"error": "Network connection failed", "code": "CONNECTION_ERROR"}
```

```
            except Exception as e:
                return {"error": f"Unexpected error: {str(e)}", "code": "UNKNOWN_ERROR"}

    except Exception as e:
        logger.error(f"Critical error in robust_stock_data: {str(e)}")
        return {"error": "Internal server error", "code": "INTERNAL_ERROR"}
```

11. Performance Optimization

Caching Strategies

```
 from functools import lru_cache
from datetime import datetime, timedelta
import asyncio

class StockDataCache:
    def __init__(self, ttl_minutes=15):
        self.cache = {}
        self.ttl = timedelta(minutes=ttl_minutes)

    def get(self, key):
        if key in self.cache:
            data, timestamp = self.cache[key]
            if datetime.now() - timestamp < self.ttl:
                return data
            else:
                del self.cache[key]
        return None

    def set(self, key, data):
        self.cache[key] = (data, datetime.now())

# Global cache instance
cache = StockDataCache()

@mcp.tool()
async def cached_stock_data(ticker: str, start_date: str, end_date: str) -> dict:
    """Stock data with caching"""
    cache_key = f"{ticker}_{start_date}_{end_date}"

    # Check cache first
    cached_result = cache.get(cache_key)
    if cached_result:
        cached_result["from_cache"] = True
        return cached_result

    # Fetch fresh data
    result = await get_stock_data(ticker, ticker, start_date, end_date)

    # Cache the result
    if "error" not in result:
        cache.set(cache_key, result)

    result["from_cache"] = False
    return result
```

12. Security Considerations

Security Best Practices

Always validate inputs and sanitize data when building production MCP servers.

Input Sanitization

```
import re
from typing import Optional

class InputValidator:
    @staticmethod
    def validate_ticker(ticker: str) -> Optional[str]:
        """Validate and sanitize ticker symbol"""
        if not ticker:
            return "Ticker cannot be empty"

        # Remove whitespace and convert to uppercase
        ticker = ticker.strip().upper()

        # Check format (letters and numbers only, 1-5 characters)
        if not re.match(r'^[A-Z0-9]{1,5}$', ticker):
            return "Invalid ticker format. Use 1-5 alphanumeric characters."

        return None  # Valid

    @staticmethod
    def validate_date(date_str: str) -> Optional[str]:
        """Validate date format"""
        if not date_str or len(date_str) != 8:
            return "Date must be in mmddyyyy format (8 digits)"

        if not date_str.isdigit():
            return "Date must contain only numbers"

        try:
            month = int(date_str[:2])
            day = int(date_str[2:4])
            year = int(date_str[4:])

            if month < 1 or month > 12:
                return "Invalid month (01-12)"
            if day < 1 or day > 31:
                return "Invalid day (01-31)"
            if year < 1900 or year > 2030:
                return "Invalid year (1900-2030)"

        except ValueError:
            return "Invalid date format"

        return None  # Valid
```

13. Testing Your MCP Server

Unit Testing

```
import unittest
from unittest.mock import patch, MagicMock
import asyncio

class TestStockServer(unittest.TestCase):
    def setUp(self):
        """Set up test fixtures"""
        self.test_ticker = "AAPL"
        self.test_start = "01012024"
        self.test_end = "12312024"

    @patch('yfinance.Ticker')
    def test_get_stock_data_success(self, mock_ticker):
        """Test successful stock data retrieval"""
        # Mock data
        mock_data = MagicMock()
        mock_data.empty = False
        mock_data.__len__.return_value = 252  # Trading days in a year
        mock_data.iloc = MagicMock()
        mock_data.iloc[0] = {'Open': 150.0}
        mock_data.iloc[-1] = {'Close': 180.0}

        mock_ticker_instance = MagicMock()
        mock_ticker_instance.history.return_value = mock_data
        mock_ticker.return_value = mock_ticker_instance

        # Test the function
        result = asyncio.run(get_stock_data(
            self.test_ticker, "Apple", self.test_start, self.test_end
        ))

        self.assertIn('ticker', result)
        self.assertEqual(result['ticker'], self.test_ticker)
        self.assertNotIn('error', result)
```

14. Deployment Considerations

Production Deployment

Deployment Checklist

1. **Environment Setup**
   - Use virtual environments
   - Pin dependency versions
   - Set up proper logging
2. **Configuration Management**
   - Use environment variables for API keys
   - Separate development/production configs
   - Implement feature flags
3. **Monitoring and Logging**
   - Log all tool calls and errors
   - Monitor response times
   - Set up alerting for failures
4. **Resource Limits**
   - Implement rate limiting
   - Set memory and CPU limits
   - Handle concurrent requests properly

15. Extending the Stock Server

Adding New Financial Tools

```
@mcp.tool()
async def get_financial_ratios(ticker: str) -> dict:
    """Calculate financial ratios for a stock"""
    try:
        stock = yf.Ticker(ticker)
        info = stock.info

        ratios = {
            "pe_ratio": info.get('forwardPE', 0),
            "pb_ratio": info.get('priceToBook', 0),
            "debt_to_equity": info.get('debtToEquity', 0),
            "profit_margin": info.get('profitMargins', 0),
            "roe": info.get('returnOnEquity', 0)
        }

        return {"ticker": ticker, "ratios": ratios}
    except Exception as e:
        return {"error": str(e)}

@mcp.tool()
async def portfolio_analysis(tickers: list[str], weights: list[float]) -> dict:
    """Analyze a portfolio of stocks"""
    if len(tickers) != len(weights):
        return {"error": "Tickers and weights must have same length"}

    if abs(sum(weights) - 1.0) > 0.001:
        return {"error": "Weights must sum to 1.0"}

    # Portfolio analysis logic here...
    return {"portfolio_stats": "Analysis complete"}
```

Real-World Integration Examples

Production Integration Ideas

- **Database Integration:** Connect to PostgreSQL or MongoDB for persistent data
- **Authentication:** Add JWT token validation for secure access
- **Rate Limiting:** Implement Redis-based rate limiting
- **Monitoring:** Add Prometheus metrics and health checks
- **Caching:** Use Redis for distributed caching

16. Complete Working Example

Here's how to use the example client to test your stock server:

```
 # Example client usage
python example_client.py

# This will automatically:
# 1. Start the MCP server
# 2. Test get_stock_data for Apple
# 3. Compare Apple vs Microsoft
# 4. Generate a stock chart
# 5. Read company information
# 6. Display all results
```

Expected Output

Sample Test Results

```
📊 MCP STOCK SERVER DEMO
==================================================

1️⃣ Listing available tools...
   🔧 get_stock_data: Get the cash balance of the given account name
   🔧 plot_stock_price: Generate stock price charts
   🔧 compare_stocks: Compare performance between two stocks

2️⃣ Getting Apple (AAPL) stock data...
   📈 Apple Inc Analysis (AAPL)
   💰 Opening Price: $170.25
   💰 Closing Price: $185.50
   📊 Price Change: $15.25
   📈 Percentage Change: 8.96%
   📊 Data Points: 252

3️⃣ Comparing Apple vs Microsoft...
   🏆 Better Performer: AAPL
   📊 Performance Difference: 2.34%
   📈 AAPL: 8.96%
   📈 MSFT: 6.62%

4️⃣ Reading Apple stock resource...
   📋 Company Information:
Company Name: Apple Inc.
Sector: Technology
Industry: Consumer Electronics
Current Price: $185.50

5️⃣ Generating Apple stock plot...
   📊 Plot generated successfully! (Base64 image data)
   📏 Data length: 15847 characters

✅ Demo completed successfully!
```

17. Conclusion

MCP servers provide a powerful way to extend AI capabilities by connecting them to external tools and data sources. The key concepts to remember:

Key Takeaways

- MCP servers act as bridges between AI and external systems
- Tools define functions that AI can call
- Resources provide read-only data access
- The chain flows: AI → MCP Tool → Business Logic → Data Source
- Proper error handling and validation are essential
- Debugging and testing utilities help ensure reliability
- Security considerations are crucial for production deployment

By following the patterns shown in this stock server example, you can build robust MCP servers for any domain-specific use case.

Next Steps

1. **Practice:** Implement the stock server and experiment with different tools
2. **Extend:** Add new financial indicators and analysis capabilities
3. **Integrate:** Connect to real databases and production APIs
4. **Deploy:** Set up monitoring and deploy to production
5. **Scale:** Implement caching, rate limiting, and load balancing

Additional Learning Resources

- **MCP Documentation:** Official protocol specification
- **FastMCP GitHub:** Examples and community contributions
- **Yahoo Finance API:** Documentation for financial data
- **Python AsyncIO:** Advanced asynchronous programming patterns

Appendices

Appendix A: Complete Stock Server Code

Full Implementation Reference

Here's the complete stock_server.py file for reference:

```
 from mcp.server.fastmcp import FastMCP
import yfinance as yf
import matplotlib.pyplot as plt
import pandas as pd
from datetime import datetime
import io
import base64

mcp = FastMCP("stock_server")

def parse_date(date_str: str) -> str:
    """Convert mmddyyyy format to yyyy-mm-dd format for yfinance"""
    try:
        month = date_str[:2]
        day = date_str[2:4]
        year = date_str[4:]
        dt = datetime(int(year), int(month), int(day))
        return dt.strftime('%Y-%m-%d')
    except (ValueError, IndexError):
        raise ValueError(f"Invalid date format: {date_str}. Expected mmddyyyy format.")

@mcp.tool()
async def get_stock_data(ticker: str, name: str, start_date: str, end_date: str) -> dict:
    """Fetch stock price data from Yahoo Finance."""
    try:
        start_formatted = parse_date(start_date)
        end_formatted = parse_date(end_date)

        stock = yf.Ticker(ticker)
        data = stock.history(start=start_formatted, end=end_formatted)

        if data.empty:
            return {"error": f"No data found for ticker {ticker}"}

        stats = {
            "ticker": ticker,
            "name": name,
            "start_date": start_formatted,
            "end_date": end_formatted,
            "data_points": len(data),
            "opening_price": round(data['Open'].iloc[0], 2),
            "closing_price": round(data['Close'].iloc[-1], 2),
            "highest_price": round(data['High'].max(), 2),
            "lowest_price": round(data['Low'].min(), 2),
            "average_price": round(data['Close'].mean(), 2),
            "price_change": round(data['Close'].iloc[-1] - data['Open'].iloc[0], 2),
            "percentage_change": round(((data['Close'].iloc[-1] / data['Open'].iloc[0]) - 1) * 100, 2)
            "average_volume": int(data['Volume'].mean())
        }
```

```
        return stats

    except Exception as e:
        return {"error": f"Failed to fetch data: {str(e)}"}

@mcp.tool()
async def plot_stock_price(ticker: str, name: str, start_date: str, end_date: str) -> str:
    """Create a plot of stock prices and return as base64 encoded image."""
    try:
        start_formatted = parse_date(start_date)
        end_formatted = parse_date(end_date)

        stock = yf.Ticker(ticker)
        data = stock.history(start=start_formatted, end=end_formatted)

        if data.empty:
            return f"No data found for ticker {ticker}"

        plt.figure(figsize=(12, 8))

        # Plot closing price
        plt.subplot(2, 1, 1)
        plt.plot(data.index, data['Close'], linewidth=2, color='blue', label='Close Price')
        plt.plot(data.index, data['Open'], linewidth=1, color='green', alpha=0.7, label='Open Price')
        plt.title(f'{name} ({ticker}) - Stock Price Chart\\n{start_formatted} to {end_formatted}',
                  fontsize=14, fontweight='bold')
        plt.ylabel('Price ($)', fontsize=12)
        plt.legend()
        plt.grid(True, alpha=0.3)

        # Plot volume
        plt.subplot(2, 1, 2)
        plt.bar(data.index, data['Volume'], alpha=0.6, color='orange', label='Volume')
        plt.title('Trading Volume', fontsize=12, fontweight='bold')
        plt.ylabel('Volume', fontsize=12)
        plt.xlabel('Date', fontsize=12)
        plt.legend()
        plt.grid(True, alpha=0.3)

        plt.tight_layout()

        # Convert plot to base64 string
        buffer = io.BytesIO()
        plt.savefig(buffer, format='png', dpi=150, bbox_inches='tight')
        buffer.seek(0)
        plot_base64 = base64.b64encode(buffer.getvalue()).decode()
        plt.close()

        return f"{plot_base64}"

    except Exception as e:
        return f"Failed to create plot: {str(e)}"

@mcp.tool()
async def compare_stocks(ticker1: str, ticker2: str, start_date: str, end_date: str) -> dict:
    """Compare two stocks over a given period."""
    try:
        stock1_data = await get_stock_data(ticker1, ticker1, start_date, end_date)
        stock2_data = await get_stock_data(ticker2, ticker2, start_date, end_date)

        if 'error' in stock1_data or 'error' in stock2_data:
            return {"error": "Failed to get data for one or both stocks"}

        comparison = {
            "comparison_period": f"{parse_date(start_date)} to {parse_date(end_date)}",
            "stock1": {
                "ticker": ticker1,
                "percentage_change": stock1_data['percentage_change'],
```

```
                "price_change": stock1_data['price_change'],
                "final_price": stock1_data['closing_price']
            },
            "stock2": {
                "ticker": ticker2,
                "percentage_change": stock2_data['percentage_change'],
                "price_change": stock2_data['price_change'],
                "final_price": stock2_data['closing_price']
            }
        }

        if stock1_data['percentage_change'] > stock2_data['percentage_change']:
            comparison['better_performer'] = ticker1
            comparison['performance_difference'] = round(
                stock1_data['percentage_change'] - stock2_data['percentage_change'], 2
            )
        else:
            comparison['better_performer'] = ticker2
            comparison['performance_difference'] = round(
                stock2_data['percentage_change'] - stock1_data['percentage_change'], 2
            )

        return comparison

    except Exception as e:
        return {"error": f"Comparison failed: {str(e)}"}

@mcp.resource("stock://data/{ticker}")
async def read_stock_resource(ticker: str) -> str:
    """Resource endpoint to get current stock information"""
    try:
        stock = yf.Ticker(ticker)
        info = stock.info

        return f"""
Stock Information for {ticker}:
Company Name: {info.get('longName', 'N/A')}
Sector: {info.get('sector', 'N/A')}
Industry: {info.get('industry', 'N/A')}
Current Price: ${info.get('currentPrice', 'N/A')}
Market Cap: ${info.get('marketCap', 'N/A'):,}
52 Week High: ${info.get('fiftyTwoWeekHigh', 'N/A')}
52 Week Low: ${info.get('fiftyTwoWeekLow', 'N/A')}
        """
    except Exception as e:
        return f"Error fetching stock resource: {str(e)}"

if __name__ == "__main__":
    mcp.run(transport='stdio')
```

Appendix B: Common Error Codes and Solutions

Error Code	Description	Common Causes	Solution
INVALID_TICKER	Ticker symbol format is incorrect	Non-existent symbol, wrong format	Use valid NYSE/NASDAQ symbols (1-5 chars)
INVALID_DATE	Date format is incorrect	Wrong format, invalid date	Use mmddyyyy format (e.g., 01152024)

NO_DATA	No data available for parameters	Market closed, delisted stock	Check ticker symbol and date range
CONNECTION_ERROR	Failed to connect to Yahoo Finance	Network issues, API unavailable	Check internet connection, retry later
RATE_LIMITED	Too many API requests	Exceeded Yahoo Finance rate limits	Implement delays, use caching
IMPORT_ERROR	Required packages not installed	Missing dependencies	Run: pip install -r requirements.txt

Appendix C: Date Format Reference

Input Format	Example	Converted To	Description	Valid Range
mmddyyyy	01152024	2024-01-15	January 15, 2024	Month: 01-12
mmddyyyy	12312023	2023-12-31	December 31, 2023	Day: 01-31
mmddyyyy	07042024	2024-07-04	July 4, 2024	Year: 1900-2030
mmddyyyy	02292024	2024-02-29	Feb 29 (leap year)	Validates leap years

Appendix D: Popular Stock Tickers for Testing

Company	Ticker	Exchange	Sector	Good for Testing
Apple Inc.	AAPL	NASDAQ	Technology	High volume, reliable data
Microsoft Corporation	MSFT	NASDAQ	Technology	Stable, good for comparisons
Google (Alphabet)	GOOGL	NASDAQ	Technology	High price, good charts
Tesla Inc.	TSLA	NASDAQ	Automotive	High volatility, interesting patterns

Amazon.com Inc.	AMZN	NASDAQ	Consumer Discretionary	Large price movements
S&P 500 ETF	SPY	NYSE	ETF	Market benchmark
NVIDIA Corporation	NVDA	NASDAQ	Technology	AI sector, high growth

Appendix E: FastMCP Server Configuration

Server Configuration Options

```
# Basic server setup
mcp = FastMCP("stock_server")

# With custom configuration
mcp = FastMCP(
    "stock_server",
    description="Stock price analysis and charting server",
    version="1.0.0"
)

# Add metadata
mcp.add_metadata("author", "Dr. Anish Roychowdhury")
mcp.add_metadata("university", "Plaksha University")

# Run with different transports
if __name__ == "__main__":
    # Standard stdio transport (default)
    mcp.run(transport='stdio')

    # HTTP transport (for web integration)
    # mcp.run(transport='http', port=8000)

    # WebSocket transport (for real-time applications)
    # mcp.run(transport='websocket', port=8001)
```

Appendix F: Development Environment Setup

Complete Setup Instructions

```
# Step 1: Create virtual environment
python -m venv mcp_stock_env
source mcp_stock_env/bin/activate  # On Windows: mcp_stock_env\\Scripts\\activate

# Step 2: Install dependencies
pip install fastmcp==0.2.0 yfinance>=0.2.28 matplotlib>=3.8.0 pandas>=2.1.0 numpy>=1.26.0

# Step 3: Create project structure
mkdir stock_mcp_project
```

```
cd stock_mcp_project
touch stock_server.py
touch example_client.py
touch requirements.txt
touch README.md

# Step 4: Add to requirements.txt
echo "fastmcp==0.2.0" >> requirements.txt
echo "yfinance>=0.2.28" >> requirements.txt
echo "matplotlib>=3.8.0" >> requirements.txt
echo "pandas>=2.1.0" >> requirements.txt
echo "numpy>=1.26.0" >> requirements.txt

# Step 5: Test installation
python -c "import yfinance; import matplotlib; import pandas; print('All packages installed successful
```

Dr. Anish Roychowdhury
Plaksha University

This comprehensive guide demonstrates MCP server development through practical, hands-on examples. The goal is to understand the underlying concepts that apply to any MCP server implementation.